A BASIC PROBLEM

- We have to store some records and perform the following:
  - Insert a new record
  - Delete a record
  - Search for a record by key
- Find a way to do these efficiently!

UNSORTED ARRAY

- Use an array to store the records, in unsorted order
  - Insert – add the records as the last entry – fast $O(1)$
  - Delete – slow at finding the target – fast at filling the hole (just take the last entry) $O(n)$
  - Search – sequential search slow $O(n)$

SORTED ARRAY

- Use an array to store the records, keeping them in sorted order
  - Insert – insert the record in proper position. Much record movement – slow $O(n)$
  - Delete – how to handle the hole after deletion? Much record movement – slow $O(n)$
  - Search – binary search – fast $O(\log n)$
**LINKED LIST**
- Store the records in a linked list (unsorted)
  - Insert – fast if one can insert node anywhere $O(1)$
  - Delete – fast at disposing the node but slow at finding the target $O(n)$
  - Search – sequential search slow $O(n)$
    - If we use a linked list, we cannot use binary search even if the list is sorted.

**MORE APPROACHES**
- Store records in a binary search tree
  - Insert – faster $O(\log n)$
  - Delete – faster $O(\log n)$
  - Search – faster $O(\log n)$

**ARRAY AS TABLE**

<table>
<thead>
<tr>
<th>studid</th>
<th>name</th>
<th>score</th>
</tr>
</thead>
<tbody>
<tr>
<td>0012345</td>
<td>andy</td>
<td>81.5</td>
</tr>
<tr>
<td>0033333</td>
<td>betty</td>
<td>90</td>
</tr>
<tr>
<td>0056789</td>
<td>david</td>
<td>56.8</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9801010</td>
<td>peter</td>
<td>20</td>
</tr>
<tr>
<td>9802020</td>
<td>mary</td>
<td>100</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9903030</td>
<td>tom</td>
<td>73</td>
</tr>
<tr>
<td>9908080</td>
<td>bill</td>
<td>49</td>
</tr>
</tbody>
</table>

Consider this problem. We want to store 1,000 student records and search them by student id.

One ‘stupid’ way is to store the records in a huge array (index 0..9999999). The index is used as the student id, i.e. the record of the student with studid 0012345 is stored at $A[12345]$
**Array as Table**
- It is also called Direct-address Hash Table.
  - Each slot, or position, corresponds to a key in \( U \).
  - If there's an element \( x \) with key \( k \), then \( T[k] \) contains a pointer to \( x \).
  - Otherwise, \( T[k] \) is empty, represented by NIL.

**Hash Function**

```
function Hash(key: KeyType): integer;
```

Imagine that we have such a magic function Hash. It maps the key (stud_id) of the 1000 records into the integers 0..999, one to one. No two different keys map to the same number.

- \( H('0012345') = 134 \)
- \( H('0033333') = 67 \)
- \( H('0056789') = 764 \)
- \( H('9908080') = 3 \)

**Hash Table**

<table>
<thead>
<tr>
<th>stud_id</th>
<th>name</th>
<th>score</th>
</tr>
</thead>
<tbody>
<tr>
<td>9908080</td>
<td>bill</td>
<td>49</td>
</tr>
<tr>
<td>0033333</td>
<td>betty</td>
<td>90</td>
</tr>
<tr>
<td>0012345</td>
<td>andy</td>
<td>81.5</td>
</tr>
<tr>
<td>0056789</td>
<td>david</td>
<td>56.8</td>
</tr>
</tbody>
</table>

To store a record, we compute Hash(stud_id) for the record and store it at the location Hash(stud_id) of the array. To search for a student, we only need to peek at the location Hash(target stud_id).
**Hash Table with Perfect Hash**

- Such magic function is called perfect hash
  - Insert: very fast $O(1)$
  - Delete: very fast $O(1)$
  - Search: very fast $O(1)$
- But it is generally difficult to design a perfect hash. (e.g., when the potential key space is large)

**How Hard Is It to Find a Good Hash Function?**

- Birthday paradox – the chance of two people in a room having the same birthday?
  - 23 people – 50%
  - 60 people – greater than 99%

**Characteristics of a Good Hash Function**

- Ideally, insertion and deletion are $O(1)$ operations
- Gives some sort of randomization in locations in the table

**Hash Function**

- A hash function maps a key to an index within in a range
- Desirable properties:
  - Simple and quick to calculate
  - Even distribution, avoid collision as much as possible

```
function Hash(key: KeyType);
```
SELECTING DIGITS

```c
unsigned int hash(string key)
{
    unsigned int hashVal = 0;
    for (int i=0; i < key.length(); i++)
        hashVal += key[i];
    return hashVal % TABLESIZE;
} // hash
```

FOLDING

```c
unsigned int hash( string key )
{
    for (i=0; i < key.length(); i++)
        hashVal = (hashVal * 128 + key[i]) % TABLESIZE;
    return hashVal;
} // hash
```

EXCLUSIVE OR

```c
unsigned int hash ( string key )
{
    unsigned int hashVal=0;
    for (i=0; i < key.length(); i++)
        hashVal = (hashVal << 5) ^ key[i] ^ hashVal;
    return hashVal % TABLESIZE;
} // hash
```

DIVISION METHOD

- $h(k) = k \mod m$
- Certain values of $m$ may not be good:
  - When $m = 2^p$ then $h(k)$ is the $p$ lower-order bits of the key.
  - Good values for $m$ are prime numbers which are not close to exact powers of 2. For example, if you want to store 2000 elements then $m=701$ ($m = \text{hash table length}$) yields a hash function:
    $$h(key) = k \mod 701$$
**COLLISION**
- Two different keys map to the same index
- For most cases, we cannot avoid collision. Therefore we must plan for them
- Probes – the number of memory locations that must be examined in order to determine the location of a key in the table

\[
\begin{align*}
H('0012345') & = 134 \\
H('0033333') & = 67 \\
H('0056789') & = 764 \\
\ldots & \\
H('9903030') & = 3 \\
H('9908080') & = 3
\end{align*}
\]

**SOLUTIONS TO COLLISION**
- The problem arises because we have two keys that hash in the same array entry, a collision. There are two ways to resolve collision:
  - **Hashing with Chaining:** every hash table entry contains a pointer to a linked list of keys that hash in the same entry
  - **Hashing with Open Addressing:** every hash table entry contains only one key. If a new key hashes to a table entry which is filled, systematically examine other table entries until you find one empty entry to place the new key

**CHAINED HASH TABLE**
- One way to handle collision is to store the collided records in a linked list. The array now stores pointers to such lists. If no key maps to a certain hash value, that array entry points to nil.

Key: 9903030  
name: tom  
score: 73

**CHAINED HASH TABLE**
- Put all elements that hash to the same slot into a linked list.
  - Slot j contains a pointer to the head of the list of all stored elements that hash to j
  - If there are no such elements, slot j contains NIL.
**CHAINED HASH TABLE**

- Hash table, where collided records are stored in linked list
  - Good hash function, appropriate hash size
    - Few collisions. Add, delete, search very fast \(O(1)\)
  - Otherwise...
    - Some hash value has a long list of collided records...
    - Insert – just insert at the head fast \(O(1)\)
    - Delete – delete from unsorted linked list slow \(O(n)\)
    - Search – sequential search slow \(O(n)\)

**OPEN ADDRESSING**

An alternative to chaining for handling collisions.

- Store all keys in the hash table itself.
- Each slot contains either a key or NIL.
- To search for key \(k\):
  - Compute \(h(k)\) and examine slot \(h(k)\).
  - Examining a slot is known as a probe.
  - If slot \(h(k)\) contains key \(k\), the search is successful.
  - If this slot contains NIL, the search is unsuccessful.
  - There’s a third possibility: slot \(h(k)\) contains a key that is not \(k\).
  - We compute the index of some other slot, based on \(k\) and on which probe (count from 0: 0th, 1st, 2nd, etc.) we’re on. Keep probing until we either find key \(k\) (successful search) or we find a slot holding NIL (unsuccessful search).

**HOW TO COMPUTE PROBE SEQUENCES**

- **Linear probing:** Given auxiliary hash function \(h\), the probe sequence starts at slot \(h(k)\) and continues sequentially through the table, wrapping after slot \(m - 1\) to slot 0. Given key \(k\) and probe number \(i (0 \leq i < m)\),
  \[ h(k, i) = (h(k) + i) \mod m. \]

- **Quadratic probing:** As in linear probing, the probe sequence starts at \(h(k)\). Unlike linear probing, it examines cells 1, 4, 9, and so on, away from the original probe point:
  \[ h(k, i) = (h(k) + c_1i + c_2i^2) \mod m \]
  (if \(c_1=0\), \(c_2=1\), it’s the example given by book)

- **Double hashing:** Use two auxiliary hash functions, \(h_1\) and \(h_2\). \(h_1\) gives the initial probe, and \(h_2\) gives the remaining probes:
  \[ h(k, i) = (h_1(k) + ih_2(k)) \mod m. \]

**OPEN ADDRESSING EXAMPLE**

- Hash( 89, 10) = 9
- Hash( 18, 10) = 8
- Hash( 49, 10) = 9
- Hash( 58, 10) = 8
- Hash( 9, 10) = 9
LINEAR PROBING: \( H(K, I) = (H(K) + I) \mod M \).

- In linear probing, collisions are resolved by sequentially scanning an array (with wraparound) until an empty cell is found.
- In following example, table size \( M = 8 \), and

<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>h</td>
<td>0, 1, 2, 3, 4, 5, 6, 7</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Action</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th># probes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Store A</td>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>Store C</td>
<td>A</td>
<td>C</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>Store D</td>
<td>A</td>
<td>C</td>
<td>D</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>Store G</td>
<td>A</td>
<td>C</td>
<td>D</td>
<td>G</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>Store P</td>
<td>A</td>
<td>P</td>
<td>C</td>
<td>D</td>
<td>G</td>
<td></td>
<td></td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>Store Q</td>
<td>A</td>
<td>P</td>
<td>C</td>
<td>D</td>
<td>G</td>
<td>Q</td>
<td>G</td>
<td></td>
<td>5</td>
</tr>
<tr>
<td>Delete P</td>
<td>A</td>
<td>#</td>
<td>C</td>
<td>D</td>
<td>Q</td>
<td>G</td>
<td></td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>Delete Q</td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
<td>Q</td>
<td></td>
<td>Q</td>
<td>G</td>
<td>5</td>
</tr>
<tr>
<td>Store E</td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
<td>Q</td>
<td></td>
<td></td>
<td>Q</td>
<td>1</td>
</tr>
<tr>
<td>Store R</td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
<td>R</td>
<td>G</td>
<td></td>
<td></td>
<td>4</td>
</tr>
<tr>
<td>Store N</td>
<td>A</td>
<td>M</td>
<td>L</td>
<td>U</td>
<td>R</td>
<td>Q</td>
<td>G</td>
<td></td>
<td>6</td>
</tr>
</tbody>
</table>

CHOOSING A HASH FUNCTION

- Notice that the insertion of Q required several probes (5). This was caused by A and P mapping to slot 0 which is beside the C and D keys.
- The performance of the hash table depends on having a hash function which evenly distributes the keys.
- Choosing a good hash function is a black art.

CLUSTERING

- Even with a good hash function, linear probing has its problems:
  - The position of the initial mapping \( i \) of key \( k \) is called the home position of \( h \).
  - When several insertions map to the same home position, they end up placed contiguously in the table. This collection of keys with the same home position is called a cluster.
  - As clusters grow, the probability that a key will map to the middle of a cluster increases, increasing the rate of the cluster’s growth.
  - **Primary clustering** – two key that hash onto different values compete for the same location in successive hashes.
  - As these clusters grow, they merge with other clusters forming even bigger clusters which grow even faster.

QUADRATIC PROBING

- Assume the value of the hash function is \( H = HASH(i) \). Cells are probed according to \( H + i^2, H + 2i^2, H + 3i^2, \ldots, H + i^2 \), where all are mod TABLESIZE.
- Will not probe every location
- Eliminates primary clustering, but not secondary clustering
**Quadratic Probing Example**

- Hash(89, 10) = 9 % 10 = 9
- Hash(89, 10) + 1² = 10 % 10 = 0
- Hash(89, 10) + 2² = 13 % 10 = 3
- Hash(89, 10) + 3² = 18 % 10 = 8
- Hash(89, 10) + 4² = 25 % 10 = 5
- Hash(89, 10) + 5² = 34 % 10 = 4
- Hash(89, 10) + 6² = 45 % 10 = 5
- Hash(89, 10) + 7² = 58 % 10 = 8
- Hash(89, 10) + 8² = 73 % 10 = 3
- Hash(89, 10) + 9² = 90 % 10 = 0

**Quadratic Probing: \( h(k, i) = (h_1(k) + C_1i + C_2i^2) \mod m \)**

Quadratic probing eliminates the primary clustering problem of linear probing by examining certain cells away from the original probe point. In the following example, table size \( m = 8 \).

<table>
<thead>
<tr>
<th>Action</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th># probes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Store A</td>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>Store C</td>
<td>A</td>
<td>C</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>Store D</td>
<td>A</td>
<td>C</td>
<td>D</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>Store G</td>
<td>A</td>
<td>C</td>
<td>G</td>
<td>G</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>Store P</td>
<td>P</td>
<td>C</td>
<td>D</td>
<td>Q</td>
<td>G</td>
<td></td>
<td></td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>Store Q</td>
<td>A</td>
<td>F</td>
<td>C</td>
<td>G</td>
<td>Q</td>
<td>G</td>
<td></td>
<td></td>
<td>3/31</td>
</tr>
<tr>
<td>Delete P</td>
<td>A</td>
<td>C</td>
<td>D</td>
<td>Q</td>
<td>G</td>
<td></td>
<td></td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>Delete Q</td>
<td>A</td>
<td>C</td>
<td>D</td>
<td>Q</td>
<td>G</td>
<td></td>
<td></td>
<td></td>
<td>3/31</td>
</tr>
<tr>
<td>Store P</td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
<td>Z</td>
<td>G</td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>Store R</td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
<td>K</td>
<td>G</td>
<td></td>
<td></td>
<td>3/41</td>
</tr>
<tr>
<td>Store Q</td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
<td>Q</td>
<td>U</td>
<td>G</td>
<td></td>
<td>3/4/0</td>
</tr>
</tbody>
</table>

**Clustering**

- Quadratic probing still has some problems
  - Not all locations are probed
  - **Secondary clustering** – when different keys that hash to the same locations compete for successive hash locations

**Double Hashing**

- **Double hashing**: Use two auxiliary hash functions, \( h_1 \) and \( h_2 \). \( h_1 \) gives the initial probe, and \( h_2 \) gives the remaining probes:
  \[
  h(k, i) = (h_1(k) + ih_2(k)) \mod m.
  \]
- Quadratic probing solves the primary clustering problem, but it has the secondary clustering problem, in which, elements that hash to the same position probe the same alternative cells. Secondary clustering is a minor theoretical blemish.
- Double hashing is a hashing technique that does not suffer from secondary clustering. A second hash function is used to drive the collision resolution.
**Double Hashing**

- Two functions
  - Step – Step(key) = 1 + key % (TABLESIZE – 2)
  - Hash – Hash(key) = key % TABLESIZE
- If TABLESIZE and TABLESIZE – 2 are prime, the function works better
- Example
  - Let key = 38 and TABLESIZE = 13

**Example Continued**

\[
\begin{align*}
1 + 38 \mod 11 &= 6 \\
38 \mod 13 &= 12 \\
(12 + 6) \mod 13 &= 5 \\
(5 + 6) \mod 13 &= 11 \\
(11 + 6) \mod 13 &= 4 \\
(4 + 6) \mod 13 &= 10 \\
(10 + 6) \mod 13 &= 3 \\
\end{align*}
\]

**Properties of a Good Hash Function**

- Minimizes collisions
- Fast to compute
- Scatter data evenly through hash table
- Uses all bits of the key
- If % (mod) is used, the size of the table should be prime

**Advantages and Disadvantages of Hashing**

- Usually a lower number of probes than with comparison searches
  - But could be bad for some data sets
- No order relationship
  - Can’t print all items in order
- Cannot look for items close to a given key